**1.Realice Ejemplos del uso de los siguientes Temas del Lenguaje Javascript en una carpeta y súbala a un repositorio en Github:**

**\*COMENTARIOS JAVASCRIPT**

JavaScript permite insertar comentarios en el código, al igual que la mayoría de los lenguajes de programación. En concreto hay dos tipos de comentarios permitidos, los comentarios en línea que comienzan con una doble barra: //, y los comentarios multilínea, que comienzan con /\* y terminan con \*/.

**\* DECLARACIONES (VAR, LET Y CONST).**

\*Una variable var declarada es vlida en todo el cuerpo de una función o en todo el programa en que fuese declarada, crea una propiedad de objeto global.

var x =6,= 6;

\*Una variable let permite declarar variables que están reservadas a un bloque de código sentencia o expresión. Si declaramos una variable en un bloque y otra en un sub-bloque, estas se consideran dos variables distintas, no crea propiedad de objeto global.

let y = 10 \* x;

\*Una variable constante se presentan en un ámbito de bloque

let y = 10 \* x;

const varna,e1 = vale1[, varname2 = value2 [, varname3 = value3]];

**\*Tipos de Datos (Boolean, Null, Undefined, Number, String, Symbol y Object), Conversion de Datos.**

**BOOLEANAS**

En JavaScript una variable es booleana si se le asigna como contenido true ó false. Estas variables se usan para almacenar información del tipo sí / no, cumple / no cumple, verdadero / falso, existe / no existe, es decir, información que únicamente puede tener dos estados.

Ejemplo: var casado = true;

**NULL**

En Javascript, al igual que en otros muchos lenguajes de programación, el valor “null” es un valor especial. El valor null es en sí mismo un valor, pero un valor que indica la ausencia de contenido, el valor vacío. Si escribimos el siguiente código:

var number1 = null;

**UNDEFINED**

Es una propiedad del objeto global, es decir, una variable de alcance global. El valor inicial de undefined es el valor primitivo [undefined](https://developer.mozilla.org/en-US/docs/Glossary/Undefined" \o "undefined: Technical review completed.). Es decir El contenido de una variable no inicializada es **undefined.**

**NUMBER**

Permite representar cualquier número que sea necesario. Para indicar que un número es negativo se precede del signo menos. Los números decimales se escriben utilizando el punto (.) como separador. Si la parte entera de un número es el cero, se admite omitir el cero. Es decir, 0.55 y .55 son ambos admitidos.

**STRING**

JavaScript permite definir el texto tanto dentro de comillas dobles como dentro de comillas simples. Por ejemplo es válido: var cadenaTexto; cadenaTexto =  "aprenderaprogramar.com";Y también es válido: cadenaTexto =  'aprenderaprogramar.com' ;

**SYMBOL**

Es un tipo de dato cuyo valor es único e inmutable pueden ser utilizados como identificadores, tiene un valor asociado de tipo string o undefined que sirve únicamente como descripción del símbolo. La descripción es un valor opcional de tipo string, solo sirve para darle la descripción l símbolo.

symbol([descripción])

**OBJET**

Es un tipo de variable especial que tiene la característica de admitir la declaración de propiedades que se asocian a ese objeto que son de lectura y escritura, se puede tanto leer el contenido de esas propiedades, como asignarles un valor. Para llamarlo se hace con el nombre del objeto.

var miObjeto= new Object();

**CONVERSION DE DATOS**

Existen dos funciones globales útiles para convertir cadenas en valores numéricos: parseInt y parseFloat.

La sintaxis básica de parseFloat es la siguiente:

var numérica = parsefloat(cadena);

La función parseFloat toma la cadena y trata de retornar un valor numérico decimal. Si la conversión no se puede completar por aparecer algún carácter extraño, devuelve el valor numérico que pueda extraer hasta la aparición de dicho valor. Si la cadena no empieza con un valor válido para un número (es decir, un +, un – ó un número, un signo . ó una letra e indicadora de exponente) la función devuelve NaN.

**\*Literales (Array, Boolean, Integer, Flotantes, Objetos, RegExp, String)**

**ARRAY**

Es una lista de cero o mas expresiones, cada uno representa un elemento array entre corchetes([]). Cuando se crea un array se inicializa con los valores especificados como sus elementos.

var cafes = [“frances”,”colombiano”,”kona”];

**BOOLEAN**

Tienen dos valores literales posibles true, false. No confundir el literal con el objeto booleano. El objeto booleano es un contenedor alrededor del tipo de dato primitivo booleano.

**INTEGER**

Pueden ser expresados en decimal (10), hexadecimal(16), octal(8) de formato. Un literal entero consiste en una secuencia de dígitos opcionalmente con el sufijo tal y como se describe sin un cero.

**FLOTANTES**

Puede tener las siguientes partes: un entero decimal, un punto decimal (“.”), una fracción, un exponente y un tipo de sufijo. Debe tener al menos un digito. Ejemplo

3.141

-3.1E12

.1e12

2E-12

**OBJETOS**

Es una lista de cero o mas parejas de nombres de propiedades asociadas con el respectivo valor, encerradas entre corchetes{}. No se debe usar un objeto literal al comienzo de una sentencia.

var vntas = “toyota”;

function tiposCarro(nombre){

if(nombre==”honda”){

return nombre;

}else{

Retur “Lo siento, nosotros no vendemos “+nombre+”.”;

}

}

**REGEXP**

compila la expresion regular una vez que el scrip a terminado de cargar, es recomendable utilizar esta representación cuando la expresión regular permanecerá sin cambios durante la ejecución de script.

Var nombrevariable = new RegExp ();

**STRING**

Corresponde a cero o mas caracteres encerrados dentro de comillas dobles o sencillas, un string debe estar delimitado por comillas del mismo tipo.

Var nombrevariable = “new RegExp”;

Var nombrevariable = ‘new RegExp’;

9.Sentencias Condicionales (if, else, switch)

**IF**

La instrucción if … else funciona de forma análoga a como lo hace en otros lenguajes de programación. Permite controlar qué procesos tienen lugar, típicamente en función del valor de una o varias variables, de un valor de cálculo o booleano, o de las decisiones del usuario. La sintaxis a emplear es:

if (condición) {

        instrucciones

} else {

        instrucciones

}

**SWITCH**

La instrucción switch es una forma de expresión de un anidamiento múltiple de instrucciones if ... else. Su uso no puede considerarse, por tanto, estrictamente necesario, puesto que siempre podrá ser sustituida por el uso de if. No obstante, a veces resulta útil al introducir eficiencia y mayor claridad en el código. Ejemplo:

switch (expresión) {

        case valor1:

        instrucciones;

        break;

        case valor2:

        instrucciones;

        break;

        default:

        sentencias;

        break;

 }

**\*Sentencias de Captura de Errores (throw, try - catch).**

**THROW**

Se utiliza la sentencia para lanzar una excepción definida por el usuario. Puede especificar un objeto cuando lanza una excepción. Se puede entonces referenciar las propiedades del objeto en el bloque catch. Ejemplo:

Throw expresión;

Throw{toString: function(){return “soy un objeto”}};

**Try-catch**

La sentencia try...catch marca un bloque de instrucciones a intentar que pueden causar alguna excepción, y declarar una o más respuestas en caso de que una excepción sea arrojada. Si una excepción es arrojada, la sentencia try...catch se encarga de atraparla.

La sentencia try...catch consiste en un bloque try, el cuál contiene una o más instrucciones, y ninguno o varios bloques catch, conteniendo sentencias que especifican que hacer si una excepción es arrojada en un bloque try. Se desea que las instrucciones dentro del bloque try se ejecuten con éxito, de caso contrario caerán en el bloque catch para ser controladas. Si ninguna instrucción dentro del bloque try (o en una función llamada dentro del bloque try) arroja una excepción, el control pasa inmediatamente al bloque catch. Si ninguna excepción es arrojada en el bloque try, el bloque catch es ignorado. Por último se ejecuta el bloque finally luego de que los bloques try y catch hayan sido ejecutados pero antes de las instrucciones que se encuentren luego de la sentencia try...catch.

**Ejemlo:**

El ejemplo llama a una función que retorna el nombre de un mes desde un arreglo basado en un valor pasado como argumento a la función. Si el valor no corresponde con el número de un mes (entre 1 y 12), una excepción es arrojada con el valor "InvalidMonthNo" y las instrucciones en el bloque catch le asignarán a la variable monthName el valor de unknown.

function getMonthName (mo) {

mo = mo-1; // Ajusta el indice del arreglo para el arreglo de meses (1=Jan, 12=Dec)

var months = ["Jan","Feb","Mar","Apr","May","Jun","Jul",

"Aug","Sep","Oct","Nov","Dec"];

if (months[mo] != null) {

return months[mo];

} else {

throw "InvalidMonthNo"; //Arroja la palabra "InvalidMonthNo" al ocurrir una excepción

}

}

try { // instrucciones a probar

monthName = getMonthName(myMonth); // La función puede arrojar una excepción

}

catch (e) {

monthName = "unknown";

logMyErrors(e); // Pasa el objeto de la excepción a un manejador de errores

}

**\*Ciclos e Iteraciones (for, while, do..while, labels para ciclos, break, continue, for..in, for..of)**

**FOR**

Un bucle for se repite hasta que la condición especificada se evalua como false.

Errores For([ExpresionInicial];[condición][expresionincremento])

\*La expresión de inicialización expresionInicial, se ejecuta si esta existe. Esta expresión habitualmente inicializa uno o mas contadores del bucle, pero la sintaxis permite una expresión con cualquier grado de complejidad. Esta expresión puede también declarar variables.

\*Se evalúa la expresión condicion. Si el valor de condicion es true, se ejecuta la sentencia del bucle. Si el valor de condicion es false, el bucle for finaliza. Si la expresión condicion es omitida, la condición es asumida como verdadera.

\*Se ejecuta la sentencia. Para ejecutar múltiples sentencias, use un bloque de sentencias ({ ... }) para agruparlas.

\*Se ejecuta la expresión expresionIncremento, si hay una, y el control vuelve al paso 2.

**WHILE**

Se ejecuta una sentencia especificada mientras cirta condición se evalua como verdadera, se ejecuta una sentencia. Cuando la condición se evalua como false, la ejecución continua como la sentencia posterior al bluble while.

Sintaxis:

while(condicion)

sentencia

Ejemplo:

n = 0;

x = 0;

while (n<3){

n++;

x + = n;

**DO.WHILE**

El bucle do … while es muy similar al bucle while. La diferencia radica en cuándo se evalúa la condición de salida del ciclo. En el bucle while esta evaluación se realiza antes de entrar al ciclo, lo que significa que el bucle puede no llegar ejecutarse. En cambio, en un bucle do … while, la evaluación se hace después de la primera ejecución del ciclo, lo que significa que el bucle obligatoriamente se ejecuta al menos en una ocasión. Después de la condición del while debemos escribir punto y coma. Por ejemplo while (contador < 10);

<!DOCTYPE html>

<html><head><title>Ejemplo aprenderaprogramar.com</title><meta charset="utf-8">

<style type="text/css">body {background-color:white; font-family: sans-serif;}

.boton{padding:15px; width: 200px;  text-align:center; clear:both;

color: white; border-radius: 40px; background: rgb(202, 60, 60);}

</style>

**<script type="text/javascript">**

function ejemploDoWhile() {

var contador = 0;var msg = '';

do { msg = msg + '\t Contando... '+ (contador+1) + '\n';

contador +=1;

} while (contador < 10);

alert ('Bucle con do ... while: \n'+ msg);

}

**</script>**

</head>

<body>

<h1>Portal web aprenderaprogramar.com</h1>

<h2>Didáctica y divulgación de la programación</h2>

<h3 class="boton" onclick="ejemploDoWhile()">Pulsa aquí</h3>

</body></html>

**LABELS**

Son simples etiquetas que nos permiten identificar sentencias o conjuntos de sentencias para poder ejecutarlos en cualquier momento en el programa. En principio podemos etiquetar cualquier sentencia, aunque, las más comunmente utilizadas como etiquetas sin ciclos (while, do/while, for y for/in).

Para crear un Label, debemos preceder a la sentencia un nombre seguido con el identificador (:). La sintaxis es la siguiente:

identificador: sentencia

Ejemplo:

Parser:

While(key != null){

Otras

**BREAK**

Incluye una etiqueta opcional que permite al programa salir de una sentencia etiquetada. La sentencia break necesita estar anidada dentro de la sentencia etiquetada. La sentencia etiquetada puede ser cualquier tipo de sentencia termina el bucle actual, sentencia switch o label y transfiere el control del programa a la sentencia de terminación de estos elementocia.

Break [etiqueta];

Ejemplo:

Function comprobarBreak(x){

Var i = 0;

While (i<6){

If (i == 3)

Break;

i++;

}return i \* x;

}

**CONTINUE**

Termina la ejecución de las sentencias de la iteración actual del bucle actual o la etiqueta y continua la ejecución del bucle con la próxima iteración.

continue [etiqueta];

Ejemplo:

i = 0

n = 0;

while (i<5){

i++;

if(i == 0)

continue;

n + = 1;

}

**FOR IN**

Esta sentencia itera sobre todas las propiedades de un objeto, en un orden arbitrario. Para cada una de las propiedades, se ejecuta la sentencia especificada.

For in itera sobre las propiedades integradas. Esto incluye todos los métodos de objetos como el indexOf() de String o el método Object de Object sin embargo, el bucle iterará sobre todas las propiedades definidas por el usuario (incluidas aquellas que sobre escriban las propiedades integradas).

Sintaxis:

for (variable in objeto)sentencia

ejemplo:

function mostrar\_Propiedades(objeto, nombreobjeto){

var resultado = “”;

for (var i in objeto){

resultado += nombreObjeto + “.” + i “=” “ + objeto[i]+ “\n”;

}

Return resultado:

}

**FOR OF**

La sentencia for...ofcrea un ciclo que itera sobre [objetos iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/iterable)s (incluyendo [Array](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array" \o "El objeto global de JavaScript Array es un constructor para matrices unidimensionales (también llamadas vectores o arreglos), que son objetos tipo lista de alto nivel.), [Map](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Map" \o "El objecto Map es un sencillo mapa clave/valor. Cualquier valor (tanto objetos como valores primitivos) pueden ser usados como clave o valor.), [Set](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Set), el objeto [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/arguments), etc), invocando una iteración personalizada con sentencias por ser ejecutadas para el valor de cada propiedad.

Ejemplo:

for (let of arr{

console.log(i);

)

**\* Funciones (Como expresiones, llamadas, alcance de variables, Recursividad, Alcance de Variables, Closures, Argumentos, Parametros, Funciones Flecha, Funciones Predefinidas).**

**Expresiones de función:**

Si bien la declaracion de la función anterior es sintácticamente una sentencia, las funciones pueden tambien ser creadas por una expresión de función. Tal función puede ser anónima; no tiene que tener un nombre. Por ejemplo, la función square podría haberse definido:

var square = function(number) {return number \* number};

var x = square(4) //x gets the value 16

**llamadas:**

Las funciones deben de estar dentro de ámbito cuando son llamadas, pero la declaración de la función puede ser izada (aparece por debajo de la llamada en el codigo), como muestra el siguiente ejemplo:

console.log(square(5));

/\* ... \*/

function square(n) { return n\*n }

Alcance de variables

**Ámbito de una FunciónEDIT**

Las variables definidas dentro de una función no pueden ser accedidas desde ningun lugar desde fuera de la función, ya que la variable se define sólo en el ámbito de la función. Sin embargo, una función puede tener acceso a todas las variables y funciones definidas dentro del ámbito en el que se define. En otras palabras, una función definida en el ámbito global puede acceder a todas las variables definidas en el ámbito global. Una función definida dentro de otra función también puede acceder a todas las variables definidas en su función padre y cualquier otra variable a la que la función padre tenga acceso.

// Las siguientes variables están definidas en el ámbito global

var num1 = 20,

num2 = 3,

nombre = "Messi";

// Esta función se define en el ámbito global

function multiplicar() {

return num1 \* num2;

}

multiplicar(); // Retorna 60

// Un ejemplo de función anidada

function obtenerPuntaje () {

var num1 = 2,

num2 = 3;

function agregar() {

return nombre + " puntaje " + (num1 + num2);

}

return agregar();

}

obtenerPuntaje(); // Retorna "Messi puntaje 5"

**Recursividad:**

La recursividad Una función puede referirse y llamarse a sí misma. Hay tres formas para una función de referirse a sí misma:

El nombre de la función

arguments.callee

una variable en el ámbito que se refiere a la función

Por ejemplo, considere la siguiente definición de función:

var foo = function bar() {

// las sentencias van aqui

};

**Closures:**

A "cierre" es una expresión (típicamente una función) que puede tener variables libres junto con un entorno que une esas variables (que "cierra" la expresión).

Los cierres son una de las características más potentes de ECMAScript (JavaScript), pero que no pueden ser explotados propiedad sin comprenderlos. Son, sin embargo, relativamente fáciles de crear, aunque sea accidentalmente, y su creación tiene consecuencias potencialmente perjudiciales, sobre todo en algunos ambientes del navegador relativamente comunes. Para evitar que accidentalmente encontrarse con los inconvenientes y tomar ventaja de los beneficios que ofrecen, es necesario entender su mecanismo. Esto depende en gran medida de la función de las cadenas de alcance en la resolución identificador y así en la resolución de nombres de propiedades de objetos.

**Argumentos:**

Usando el objeto argumentsEDIT

Los argumentos de una función son mantenidos en un objeto similar a un array. Dentro de una función, los argumentos pasados a la misma pueden ser referenciados de la siguiente forma:

arguments[i]

donde i es el número ordinal del argumento, con base cero. Entonces, el primer argumento pasado a una función sería arguments[0]. El número total de argumentos es retornado por arguments.length.

**Parametros:**

Parametros de funcionEDIT

Comenzando con ECMAScript 6, hay dos nuevos tipos de parametros: Parametros por defecto y parametros en ristra.

Parametro por defecto: En JavaScript, los parametros de funciones estan establecidos por defecto a undefined. Sin embargo, En ciertas situaciones puede ser util establecerlos a un valor por defecto diferente. Es entonces cuando los parametros por defecto pueden ayudar.

En el pasado, la estrategia general para establecer los parametros por defecto era comprobar los valores de estos en el cuerpo de la funcion y asignar un valor si estos eran undefined. Si en el siguiente ejemplo , no se suministra ninngun valor de llamada, Su valor seria undefined cuando se evalue a\*b y multiply haria tener retornado NaN. Sin embargo, esto se captura con la linea segunda en este ejemplo:

function multiply(a, b) {

b = typeof b !== 'undefined' ? b : 1;

return a\*b;

}

multiply(5); // 5

**FunFunciones flechaEDIT**

Una expresion de funcion flecha (tambien conocida como funcion flecha gorda o fat arrow function en ingles) tiene una sintaxis más corta comparada con las expresiones de función y lexicamente ata el valor de this. Las funciones flecha son siempre funciones anonimas. Vease tambien esta entrada en el blog hacks.mozilla.org : "ES6 In Depth: Arrow functions" (en ingles).

Dos factores influenciaron la introdución de las funciones flecha: funciones más cortas y this lexico (que feo suena eso ;) ).

Funciones más cortas: En algunos patrones funcionales, funciones mas cortas son bienvenidas. Compare:

var a = [

"Hydrogen",

"Helium",

"Lithium",

"Beryl­lium"

];

var a2 = a.map(function(s){ return s.length });

var a3 = a.map( s => s.length );ciones Flecha:

Funciones Predefinidas:

**Funciones predefinidasEDIT**

JavaScript tiene varias funciones predefinidas de nivel superior:

eval

isFinite

isNaN

parseInt and parseFloat

Number and String

encodeURI, decodeURI, encodeURIComponent, and decodeURIComponent (all available with Javascript 1.5 and later).

Las siguientes secciones presentan estas funciones. Vea la Refencia JavaScript para obtener información detallada sobre todas estas funciones.

**Funcion eval:**

La función eval evalúa una cadena de código JavaScript sin hacer referencia a un objeto en particular. La sintaxis de eval es:

eval(expr);

donde expr es una cadena a ser evaluada.

**\* Operadores (Asignacion y Asignacion destructurada, Comparacion, Aritmeticos, Logicos, String, Ternario, Delete, typeof, void, in, instanceOf, presendencia de operadores, this, super, operador de propagación**

**Operadores de asignación**

Un operador de asignación asigna un valor a el operador de la izquierda en función a el valor del operando de la derecha. El operador básico de asignación es el igual (=), que asigna el valor de la derecha a el operador de la izquierda. Por ejemplo, x = y, está asignando el valor y a x.

Los otros operadores de asignación y su forma abreviada están en la siguiente tabla:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Operador abreviado** | **Significado** | | x += y | x = x + y | | x -= y | x = x - y | | x \*= y | x = x \* y | | x /= y | x = x / y | | x %= y | x = x % y | | x <<= y | x = x << y | | x >>= y | x = x >> y | | x >>>= y | x = x >>> y | | x &= y | x = x & y | | x ^= y | x = x ^ y | |

**Operadores de comparación**

Un operador de comperación compara sus operandos y devuelve un valor lógico basado en si la comparación es verdadera (true) o falsa (false). Los operadores pueden ser númericos, cadena de caracteres (strings), lógico u objetos. Los Strings son comparados banadose en un orden lexicográfico estandar, basandose en Unicode. En la mayoría de los casos, si dos operadores no son del mismo tipo, JavaScript intenta convertirlos en el tipo apropiado para permiter la comparación, generalmente, convirtiendolos a tipo númerico. Las únicas excepciones que tiene esta conversión son los operadores === y !== que implican que ambos valores sean del mismo tipo a la hora de la comparación. Estos operadores no intentan convertir los operandos a tipo compatible antes de comprobar su igualdad. La siguiente tabla describe los operadores de comparación en base a el siguiente código de ejemplo:

var var1 = 3, var2 = 4;

|  |  |  |
| --- | --- | --- |
| **Operador** | **Descripción** | **Ejemplos devolviendo true** |
| Igualdad (==) | Devuelve true si ambos operandos son iguales. | 3 == var1  "3" == var1  3 == '3' |
| Desigualdad (!=) | Devuelve true si ambos operandos no son iguales. | var1 != 4 var2 != "3" |
| Estrictamente igual (===) | Devuelve true si los operandos son igual y tienen el mismo tipo. Mira también [Object.is](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) y [sameness in JS](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Sameness" \o "/en-US/docs/Web/JavaScript/Guide/Sameness). | 3 === var1 |
| Estrictamente desiguales(!==) | Devuelve true si los operandos no son igual y/o no tienen el mismo tipo. | var1 !== "3" 3 !== '3' |
| Mayor que (>) | Devuelve true si el operando de la izquierda es mas grande que el operando de la derecha. | var2 > var1 "12" > 2 |
| Mayor o igual que (>=) | Devuelve true si el operando de la izquierda es mas grande que el operando de la derecha. | var2 >= var1 var1 >= 3 |
| Menor que (<) | Devuelve true si el operando de la izquierda es mas pequeño que el operando de la derecha. | var1 < var2 "2" < "12" |
| Menor o igual que (<=) | Devuelve true si el operando de la izquierda es mas pequeño o igual que el operando de la derecha. | var1 <= var2 var2 <= 5 |

**Operadores aritméticos**

Los operadores aritméticos toman valores númericos (tanto literales como variables), y sus operandos y devuelven un único resultado numerico. Los operadores aritméticos estandar son la suma (+), la multiplicación(\*), la resta(-) y la división (/). Estos operadores funcionan como en los otros lenguajes de programación cuando son usados con números de puntos flotantes (en particular, tenga en cuenta que la división por cero produce[Infinity](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Infinity)). Por ejemplo:

console.log(1 / 2); /\* devuelve 0.5 \*/

console.log(1 / 2 == 1.0 / 2.0); /\* devuelve true \*/

| **Operador** | **Descripcion** | **Ejemplo** |
| --- | --- | --- |
| % (Modulo) | Operador binario. Devuelve el resto de la división de dos operandos | 12 % 5 returns 2. |
| ++ (Incremento) | Operador unario. Suma uno a el operando. Si es usado antes del operando (++x) devuelve el valor del operando después de añadirle 1 y si se usa después del operando (x++) devuelve el valor de este operando antes de añadir 1. | Si x es 3, entonces ++xestablece x a 4 y devuelve 4, mientras que x++ devuelve 3 y, solo después de devolver el valor, establece x a 4. |
| -- (Decrement) | Operador unario. Resta one al operando. El valor devuelto es igual que para el operador de incremento | Si x es 3, entonces --xestablece x a 2 y devuelve 2, mientras que x-- devuelve 3 y, solo después de devolver el valor, establece x a 2. |
| - (Unary negation) | Operación unaria. Devuelve la negación del operando | Si x es 3, entonces -xdevuelve -3. |

**Operadores lógicos**

Los operadores lógicos son típicamente utilizado con valores boléanos (valores lógicos); ellos devuelven un valor boleano. Sin embargo, los operadores && y || devuelven el valor de uno de los operados, así que si estos operadores son usados con valores no boléanos, devolverán un valor no boleano. Los operadores lógicos están descritos en la siguiente tabla:

| **Operador** | **Uso** | **Descripcion** |
| --- | --- | --- |
| && | expr1 && expr2 | (Logical AND) Devuelve expr1 si puede ser convertido a false o sino devuelve expr2. Así, cuando se usa con valores boleanos, && devuelve true si ambos operandos son true, en caso contrario devuelve false. |
| || | expr1 || expr2 | (Logical OR) Devuelve expr1 si puede ser convertido a false o sino devuelve expr2. Así, cuando se usa con valores boleanos, && devuelve true si cualquiero operandos es true, o false si ambos son falsos. |
| ! | !expr | (Logical NOT) Devuelve false si el operando no puede ser convertido a true, en caso contrario, devuelve true |

**Operadores de cadena de caracteres**

Además de los operadores de comparación, que pueden ser usado para evaluar strings, el operador de concatenación (+) une dos valores string, devolviendo otro string que es la unión de estos. Por ejemplo,  "Mi" + "String" devuelve un único string "Mi string".

La versión acortada de este operador de asignación += puede ser usado también para concatenar string. Por ejemplo, si una variable myString tiene el valor "alpha", el resultado de aplicar la operacion myString += "bet" asigna el string "alphabet" a esta variable.

**Ejemplo:**

Una cadena es, la mayoría de las veces, texto, por ejemplo “Hola Mundo!”. Para unir dos o más variables de cadena, se usa el operador +

txt1="Hoy es un"

txt2="hermoso día!"

txt3=txt1+txt2

**Operador ternario**

Es aquel que nos sirve para escribir el código de una sentencia if de una forma más sencilla. Imaginemos que tenemos un ejemplo que nos calcula cuál es el mayor de dos número en [Javascript](http://www.manualweb.net/tutorial-javascript/" \o "Manual de Javascript). El código sería sencillo como comprobamos a continuación.

var x = 10;

var y = 8;

if (x>y)

document.write("El mayor de los dos números es " + x);

else

document.write("El mayor de los dos números es " + y);

**Sintaxis:**

Resultado = (condicion)?valor1:valor2;

**Operadores Unarios :** Una operación unaria es una operación con un único operando.

|  |  |  |
| --- | --- | --- |
| [**delete**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/delete) | Definicion:  El operador delete elimina una propiedad de un objeto.  Sintaxis:  delete objectName;  delete objectName.property;  delete objectName[index];  delete property; | Ejemplo:  delete objeto.propiedad  delete objeto['propiedad'] |
| [**void**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/void) | El operador void descarta el valor de retorno de una expresión.  El voidoperador especifica una expresión a ser evaluada sin devolver un valor. expressiones una expresión JavaScript de evaluar. Los paréntesis que rodean la expresión son opcionales, pero es un buen estilo para usarlos.  Sintaxis:  void (expression)  void expression | El siguiente código crea un enlace de hipertexto que envía un formulario cuando el usuario hace click en él.  <a href="javascript:void(document.form.submit())">  Haga click aquí para enviar</a> |
| [**typeof**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/typeof) | El operador typeof determina el tipo de un objeto dado.  El typeofoperador devuelve una cadena que indica el tipo del operando sin evaluar. operandes la cadena, variables, palabra clave o un objeto para el que el tipo ha de ser devuelto. Los paréntesis son opcionales.  Sintaxis:  typeof operand  typeof (operand) | En el ejemplo siguiente se prueba el tipo de datos de las variables.  JavaScript  var index = 5;  var result = (typeof index === 'number');  // Output: true  var description = "abc";  var result = (typeof description === 'string');  // Output: true |

#### **Operador in**

El operador in devuelve true si la propiedad especificada está en el objeto especificado. La sintaxis es:

propNameOrNumber in objectName

Donde propNameOrNumber es una cadena o expresión numérica que representa un nombre de propiedad o índice de matriz y objectName es el nombre de un objeto.

**Operador instanceof**

El operador instanceof devuelve true si el objeto especificado es del tipo de objeto especificado. La sintaxis es:

objectName instanceof objectType

Donde objectName es el nombre del objeto a comparar con objectType y objectType es un tipo de objeto, como Date o Array.

Utilice instanceof cuando necesite confirmar el tipo de un objeto en tiempo de ejecución. Por ejemplo, al captar excepciones, puede derivarse a un código de manejo de excepciones diferente dependiendo del tipo de excepción lanzada.

#### **Operador this**

Usar la thispalabra clave para referirse al objeto actual. En general, thisse refiere al objeto que llama en un método. Utilice thiscomo sigue:

**this["propertyName"]**

**this.propertyName**

**\* Colecciones (Array, Metodos Array, Matrices, Maps, Sets)**

|  |  |  |
| --- | --- | --- |
| **Colecciones** | **Definición** | **Ejemplo** |
| **Array** | Sirve Para tener acceso a los miembros de una colección mediante un índice.  Array es un constructor para matrices unidimensionales (también llamadas vectores o arreglos), que son objetos tipo lista de alto nivel. | var fruits = ["Manzana", "Plátano"];  console.log(fruits.length); |
| **Metodos Array** | [**Array.from()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/from)  Crea un nuevo Array de un objeto iterable o parecido a un array  [**Array.isArray()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/isArray)  Devuelve true si una variable es un Array, de lo contrario devuelve false.  [**Array.observe()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/observe)  Observa los cambios del Array asincronicamente, similar a [Object.observe()](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Object/observe" \o "El método Object.observe() es usado para observar de forma asíncrona cambios sobre un objeto.  Este método transmite información sobre cambios en el objeto, en el orden en que estos ocurren.) para los objetos. Provee un flujo de cambio en orden.  [**Array.of()**](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Array/of)  Crea un nuevo Array con un numero de argumentos variable, sin importar numero o tipo de argumentos. | function f() {  return Array.from(arguments);  }  f(1, 2, 3);  var s = new Set(["foo", window]);  Array.from(s);  var m = new Map([[1, 2], [2, 4], [4, 8]]);  Array.from(m);  Array.from("foo");  Array.from([1, 2, 3], x => x + x);  Array.from({length: 5}, (v, k) => k);  Array.isArray([]);  Array.isArray([1]);  Array.isArray(new Array());  Array.isArray(Array.prototype);  Array.isArray();  Array.isArray({});  Array.isArray(null);  Array.isArray(undefined);  Array.isArray(17);  Array.isArray('Array');  Array.isArray(true);  Array.isArray(false);  Array.isArray({ \_\_proto\_\_: Array.prototype });  var arr = ['a', 'b', 'c'];  Array.observe(arr, function(changes) {  console.log(changes);  });  arr[1] = 'B';  arr[3] = 'd';  arr.splice(1, 2, 'beta', 'gamma', 'delta');    Array.of(1); // [1]  Array.of(1, 2, 3); // [1, 2, 3]  Array.of(undefined); // [undefined] |
| **Matrices** | Estas matrices son dispersas  las matrices no tienen las propiedades y los métodos de un objeto. Para direccionar matrices se usan corchetes ([]) | Los corchetes rodean un valor numérico o una expresión que se evalúa como un número entero.  var entryNum = 5;  sample = new Array();  sample[1] = "Maple Street";  sample[entryNum] = 25;  document.write (sample[1]);  document.write (" ");  document.write (sample[entryNum]); |
| **Maps** | Almacenan pares de clave-valor y permiten agregar, recuperar y quitar miembros mediante la clave. La clave y el valor pueden ser de cualquier tipo. Este objeto permite enumerar los miembros de la colección utilizando el método **forEach** y comprobar el tamaño de la colección utilizando el método **size** | En este ejemplo, se tiene acceso a los miembros mediante get y forEach  var m = new Map();  m.set(1, "black");  m.set(2, "red");  m.set("colors", 2);  m.set({x:1}, 3);  document.write(m.get(2));  document.write("<br />");  m.forEach(function (value, key, mapObj) {      document.write(item.toString() + "<br />");  }); |
| **Sets** | El objeto **Set** almacena valores de cualquier tipo.  Este objeto permite enumerar los miembros de la colección utilizando el método **forEach** y comprobar el tamaño de la colección utilizando el método **size** | En este ejemplo, la función de devolución de llamada toma un parámetro, que es el valor del elemento actual de la colección.  var s = new Set();  s.add("Thomas Jefferson");  s.add(1776);  s.add("founding father");  s.forEach(function (value) {      document.write(item.toString() + ", ");  }); |

**\* Objetos (Declaracion, Propiedades, Funciones de Listado, Constructores, Create, Metodos, Herencia, getters y setters, Comparacion de Objetos).**

La programación orientada a objetos es un paradigma de programación que utiliza la abstracción para crear modelos basados ​​en el mundo real

**\* Propiedades**

Las propiedades son variables contenidas en la clase, cada instancia del objeto tiene dichas propiedades. Las propiedades deben establecerse a la propiedad prototipo de la clase (función), para que la herencia funcione correctamente. Para trabajar con propiedades dentro de la clase se utiliza la palabra reservada **this** , que se refiere al objeto actual. El acceso (lectura o escritura) a una propiedad desde fuera de la clase se hace con la sintaxis: **NombreDeLaInstancia.**

**Ejemplo:**

En el siguiente ejemplo definimos la propiedad **primerNombre**de la clase Persona y la definimos en la creación de la instancia

function Persona(primerNombre) {

this.primerNombre = primerNombre;

alert('Una instancia de Persona');

}

var persona1 = new Persona("Milena");

var persona2 = new Persona("Joaco");

// Muestra el primer nombre de persona1

alert ('persona1 es ' + persona1.primerNombre); // muestra "persona1 es Milena"

alert ('persona2 es ' + persona2.primerNombre); // muestra "persona2 es Joaco"

\* **Funciones de Listado**

Para definir un tipo de objeto, puedes crear una función del tipo de objeto que especifica su nombre, propiedades y métodos.

**Ejemplo:**

supongamos que deseas crear un tipo de objeto para los coches. Quieres que este tipo de objeto se llame auto, y deseas que tenga propiedades de marca, modelo y el año. Para ello, podrías escribir la siguiente función:

function Auto(marca, modelo, annio) {

this.marca = marca;

this.modelo = modelo;

this.annio = annio;}

Ejemplo: Un objeto puede tener una propiedad que es en sí mismo otro objeto. Por ejemplo, supon que defines un objeto llamado persona de la siguiente manera:

function Persona(nombre, edad, sexo) {

this.nombre = nombre;

this.edad = edad;

this.sexo = sexo;

}

\* **Constructores**

El constructor se usa para establecer las propiedades del objeto o para llamar a los métodos para preparar el objeto para su uso. Más adelante describiremos como agregar métodos a clase y sus definiciones ya que se realiza utilizando una sintaxis diferente.

El constructor es llamado en el momento de la creación de la instancia (el momento en que se crea la instancia del objeto). El constructor es un método de la clase. En JavaScript, la función sirve como el constructor del objeto, por lo tanto, no hay necesidad de definir explícitamente un método constructor.Cada acción declarada en la clase es ejecutada en el momento de la creación de la instancia.

Ejemplo.

En el siguiente ejemplo, el constructor de la clase Persona muestra un alerta que dice (**Una instancia de persona)** cuando se crea la instancia de la clase Persona.

function Persona() {

alert('Una instancia de Persona');

}

var persona1 = new Persona();

var persona2 = new Persona();

**\*** **Create**

El **Object.create()**método crea un nuevo objeto con el objeto prototipo especificada y las propiedades.

**Sintaxis**

Object.create ( proto [, propertiesObject ])

### parámetros

**proto**

El objeto que debe ser el prototipo del objeto recién creado.

**PropertiesObject** :Opcional. Si se especifica, no [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined" \o "La propiedad no definida mundial representa el valor simple sin definir.  Es uno de los tipos primitivos de JavaScript.), un objeto cuyas propiedades enumerables propia (es decir, aquellas propiedades que se definen a sí misma y no en propiedades enumerables lo largo de su cadena de prototipo) especifican descriptores de propiedades que se añade al objeto de nueva creación, con los nombres de las propiedades correspondientes. Estas propiedades se corresponden con el segundo argumento de [Object.defineProperties()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties" \o "Los Object.defineProperties () método se establecen nuevas propiedades existentes o modifica directamente sobre un objeto, la devolución del bien.).

Ejemplo:

function crearObjeto(proto) {

function ctor() { }

ctor.prototype = proto;

return new ctor();

}

// uso:

Estudiante.prototype = crearObjeto(Persona.prototype);

#### **Métodos**

los métodos son objetos como lo es una función normal y se vinculan a un objeto como lo hace una propiedad, lo que significa que se pueden invocar desde "fuera de su contexto".

Los métodos siguen la misma lógica que las propiedades, la diferencia es que son funciones y se definen como funciones. Llamar a un método es similar a acceder a una propiedad, pero se agrega () al final del nombre del método, posiblemente con argumentos.

**Ejemplo:** En el siguiente ejemplo se define y utiliza el método diHola() para la clase Persona.

function Persona(primerNombre) {

this.primerNombre = primerNombre;

}

Persona.prototype.diHola = function() {

alert ('Hola, Soy ' + this.primerNombre);

};

var persona1 = new Persona("Milena");

var persona2 = new Persona("Joaco");

// Llamadas al método diHola de la clase Persona.

**\* Herencia**

La herencia es una manera de crear una clase como una versión especializada de una o más clases (JavaScript sólo permite herencia simple). La clase especializada comúnmente se llama hija o secundaria, y la otra clase se le llama padre o primaria. En JavaScript la herencia se logra mediante la asignación de una instancia de la clase primaria a  la clase secundaria, y luego se hace la especialización.

**\*Getters y Setters**

Puede definir **getters** y **setters** de cualquier objeto predefinido del núcleo o de un objeto definido por el usuario que admita la adición de nuevas características.

|  |  |  |
| --- | --- | --- |
| **Concepto**    **getter**  **Setter**    **getter**  **Setter**  **pueden añadirse a un objeto** | **Definición**  Un **getter** es un método que obtiene el valor de una propiedad específica.  Un **setter** es un método que establece el valor de una propiedad específica  Cuando se define getters y setters utilizando los [inicializadores de objeto](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#Using_object_initializers) todo lo que tienes que hacer es anteponer un método getter con get y un método setter con set. Por supuesto, el método getter no debe esperar un parámetro, mientras que el método setter espera exactamente un parámetro (el nuevo valor a definir). Por ejemplo:  var o = {  a: 7,  get b() { return this.a + 1; },  set c(x) { this.a = x / 2; }  };  \*Los getters y setters también pueden añadirse a un objeto en cualquier momento después de su creación utilizando el método Object.defineProperties. El primer parámetro de este método es el objeto sobre el que se quiere definir el getter o setter.  El segundo parámetro es un objeto cuyo nombre de propiedad son los nombres getter o setter, y cuyos valores de propiedad son objetos para la definición de las funciones getter o setter. | Ejemplo:  La siguiente  ilustracion muestra como getters y setters podrían trabajar para un objeto o definido por el usuario  js> var o = {a: 7, get b() {return this.a + 1;}, set c(x) {this.a = x / 2}};  [object Object]  js> o.a;  7  js> o.b;  8  js> o.c = 50;  js> o.a;  25  \*Estas declaraciones definen un getter y setter para la propiedad de year:  js> var d = Date.prototype;  js> Object.defineProperty(d, "year", {  get: function() {return this.getFullYear() },  set: function(y) { this.setFullYear(y) }  });  \* Estas declaraciones utilizan el getter y setter de un objeto Date:  js> var now = new Date;  js> print(now.year);  2000  js> now.year = 2001;  987617605170  js> print(now);  Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001  Ejemplo  var o = { a:0 }  Object.defineProperties(o, {  "b": { get: function () { return this.a + 1; } },  "c": { set: function (x) { this.a = x / 2; } }  });  o.c = 10 // Ejecuta el setter, que asigna 10/2 (5) a la propiedad 'a'  console.log(o.b) // Ejecuta el getter, que produce a + 1 o 6 |

**\*Comparacion de Objetos**

Cuando comparamos dos objetos que hacen referencia al mismo objeto y cuando lo comparamos devolverá true. Comparando dos objetos que se ven exactamente igual, me refiero a ambos objetos que tienen los mismos métodos y propiedades devolverá false.

**Ejemplo:**

// variable de referencia del objeto fruta

var fruta = {nombre: "manzana"};

// variable de referencia del objeto fructificar

var fructificar = {nombre: "manzana"};

fruta == fructificar // retorna false

fruta === fructificar // retorna false

Nota: El operador "===" se utiliza para comprobar el valor así como el tipo, ejemplo:   
1 === "1"  // retorna false  
1 == "1"   // retorna true

//variable de referencia del objeto fruta

var fruta = {nombre: "manzana"};

// variable de referencia del objeto fructificar

var fructificar = fruta; // asignamos la referencia del objeto fruta a la variable de referencia del objeto fructificar

// aquí fruta y fructificar apuntan al mismo objeto llamado fruta

fruta == fructificar // retorna true

// aquí fruta y fructificar apuntan al mismo objeto llamado fruta

fruta === fructificar // retorna true